METHOD AND SYSTEM FOR PASSING MESSAGES BETWEEN THREADS

TECHNICAL FIELD

This invention relates generally to message passing between threads, and, more particularly, to passing messages between threads using a message queue.

BACKGROUND OF THE INVENTION

There are currently many schemes for allowing different threads of execution on a computer to communicate with one another. Typically, these schemes involve using the message passing mechanisms of the computer's operating system. However, there are many situations in which it is not desirable to call on these mechanisms from directly within a program. An example of such a situation is when the threads that wish to communicate are scripting threads. Scripting threads are simply threads that execute according to a scripting language, such as JavaScript or PERL. Programmers often prefer to use scripting languages rather than compiled languages because scripting languages hide a lot of the underlying complexity of the machines on which they run. This complexity is, instead, handled by the "script engine," which is the name often given to the program that interprets the script. If a script programmer is forced to rely explicitly on the message passing facilities of the operating system, then he runs the risk of reintroducing some of the complexity that he sought to avoid by choosing a script language in the first place.

Thus it can be seen that there is a need for a method and system for passing messages between threads that avoids the above-mentioned disadvantages.

SUMMARY OF THE INVENTION

In accordance with this need, a method and system for passing messages between threads is provided. According to the method and system, a sending thread interprets a block of source and, according to the source code, communicates with a receiving thread by passing a reference to the message to a message queue associated with the receiving thread. The reference may be passed without explicitly invoking the inter-process or inter-thread message passing services of the computer's operating system from within the block of source code. The sending thread may also have a message queue associated with it, and the sending thread's queue may include a reference to the receiving thread's queue. The sending thread can use this reference to pass messages to the receiving thread's queue.

Additional features and advantages of the invention will be made apparent from the following detailed description of illustrative embodiments that proceeds with reference to the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

While the appended claims set forth the features of the present invention with particularity, the invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

FIGURE 1 is an example of a network;

FIG. 2 is an example of a computer;

FIG. 3 is an example of a software architecture;

FIG. 4 is an example of steps that may be followed for passing a message from one thread to another; and,

FIG. 5 is an example of a system for compiling a program.

DETAILED DESCRIPTION OF THE INVENTION

The invention is generally directed to a method and system for passing messages asynchronously from one thread to another without explicitly invoking the message passing mechanisms of an operating system from within the source code of the sending thread. In an embodiment of the invention, the receiving thread has a queue for holding messages and the sending thread has the address (in the form of a "reference") of the receiving thread's queue. The sending thread passes, by reference, the message to the receiving thread's queue using this address. The invention may be used in conjunction with scripting threads.

Although it is not required, the invention may be implemented by computer-executable instructions, such as program modules, that are executed by a computer. Generally, program modules include routines, programs, objects, components, data structures and the like that perform particular tasks or implement particular abstract data types. On many computers, modules execute within an address space of the computer's memory, which is typically defined as a "process." The point of execution of the program instructions is often referred to as a "thread." As is conventional, multiple threads of execution may exist for a single program in a process. Multiple processes may be executed on a single machine, with each process

having one or more threads of execution. Thus, when communication between threads is discussed herein, it may mean communication between threads in a single process or communication between threads in different processes.

In this description, reference will be made to one or more "objects" performing functions on a computer. An "object" is a programming unit used in many modern programming languages. Objects may also execute on a computer as part of a process, procedure, and may be manifested as executable code, a DLL, an applet, native instruction, module, thread, or the like.

The invention may be implemented on a variety of types of computers, including personal computers (PCs), hand-held devices, multi-processor systems, microprocessor-based on programmable consumer electronics, network PCs, minicomputers, mainframe computers and the like. The invention may also be employed in distributed computing environments, where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, modules may be located in both local and remote memory storage devices.

An example of a networked environment in which this system may be used will now be described with reference to FIG. 1. The example network includes several computers 100 communicating with one another over a network 102, represented by a cloud. Network 102 may include many well-known components, such as routers, gateways, hubs, etc. and may allow the computers 100 to communicate via wired and/or wireless media.

Referring to FIG. 2, an example of a basic configuration for a computer on which the system described herein may be implemented is shown. In its most basic configuration, the computer 100 typically includes at least one processing unit 112 and memory 114. Depending on the exact configuration and type of the computer 100, the memory 114 may be volatile (such as RAM), non-volatile (such as ROM or flash memory) or some combination of the two. This most basic configuration is illustrated in FIG. 2 by dashed line 106. Additionally, the computer may also have additional features/functionality. For example, computer 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or tape. Computer storage media includes volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules, or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disk (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to stored the desired information and which can be accessed by the computer 100. Any such computer storage media may be part of computer 100.

Computer 100 may also contain communications connections that allow the device to communicate with other devices. A communication connection is an example of a communication medium. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a

modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. The term computer readable media as used herein includes both storage media and communication media.

Computer 100 may also have input devices such as a keyboard, mouse, pen, voice input device, touch input device, etc. Output devices such as a display 116, speakers, a printer, etc. may also be included. All these devices are well known in the art and need not be discussed at length here.

The technology described herein may be implemented in a variety of ways. FIG. 3 shows an architecture that may be used in the context of scripting threads. Its basic components include a first instance 153 of a script engine executing in a first thread 122, and a second instance 154 of the script engine executing in a second thread 124. The first and second instances will hereinafter be referred to as script engine 153 and script engine 154, respectively. As discussed in the background section, a script engine is a program that interprets script and executes operations on a computer according to the script. Implementations of a script engine include, but are not limited to, a JScript engine, a VBScript engine, and a PERL engine. The basic components further include a queue 126 that exists within the thread 122, and a queue 128 that exists within the thread 124.

Threads 122 and 124 may be executing on the same machine or on separate machines in communication with one another. To execute thread 122, the script

engine 153 interprets and acts upon the block of script 122a. To execute thread 124, the script engine 154 interprets and acts upon block of script 124a. The script engines 153 and 154 interpret the text of the script blocks 122a and 124a in a well-known manner. For the purposes of this description, the script engines 153 and 154 are assumed to be instances of a JScript engine, and the script blocks 122a and 124a are assumed to be blocks of JScript. Any suitable type of scripting engine and scripting language may be used, however.

Queue 126 includes the following objects: an array 130 of messages received from thread 124 and not yet processed by thread 122; a string data structure 132 containing the name for the queue 126; a string data structure 134 containing the name of the signal that tells the queue 126 when a message has arrived; a reference variable 136 containing the address of the queue 128; a low index 138 containing the address of the oldest message in the queue 126; and a high index 140 containing the address of the message most recently added to the queue 126.

Queue 128 includes the following objects: an array 142 of messages received from thread 122 and not yet processed by thread 124; a string data structure 144 containing the name for the queue 128; a string data structure 146 containing the name of the signal that tells the queue 128 when a message has arrived; a reference variable 148 containing the address of the queue 126; a low index 150 containing the address of the oldest message in the queue 128; and a high index 152 containing the address of the message most recently added to the queue 128.

To initialize the above-described data structures according to an embodiment of the invention, one of the threads creates its own queue object and then sends the

address of this object (in the form of a reference) to the second thread. The second thread then creates its own queue object and completes the initialization by cross-referencing its queue with the queue of the first thread. For example, in the embodiment illustrated in FIG. 3, references can be passed from thread 122 to thread 124 via the IDispatch interface 157 of a COM object 159 that is defined within the script engine 153. Similarly, the script engine 154 can pass a reference from thread 124 to thread 122 via the IDispatch interface 156 of a COM object 158 that is defined within the script engine 154. Accordingly, the references 136 and 148 referred to above may, for example, be implemented by the script engine 154 as IDispatch COM interfaces. COM objects are platform-independent, and the IDispatch interface allows references to objects to be passed across thread, process and machine boundaries. Any other cross-platform object or mechanism may be used, however.

For one thread to send a message to another thread according to an embodiment of the invention, the sending thread creates a message object, and inserts the message – which includes a string of characters – into the message object. The sending thread then passes a reference (via the IDispatch interface, for example) to the message object to the message queue of the receiving thread. The sending thread then increments the high index of the receiving thread's queue. The sending thread will already have a reference to the receiving thread's queue, having acquired the reference during an initialization procedure, such as the one described above. Once the message is sent, the sending thread need not keep the reference to the message. If, however, the sending thread wishes to wait until it receives a reply to the messages, then it is preferable that the sending thread maintains the reference to the message.

The sending thread may then periodically check the sent message to see if a "reply" flag has been set.

To check for the presence of a new message according to an embodiment of the invention, the receiving thread compares the low index and the high index of the queue. If they are not equal, then there is at least one message in the queue. The receiving thread may retrieve a received message, reply if necessary, delete the reference to the message, and increment the low index. To reply to the message, the receiving thread flags the message as "replied," and sends the message back to the original sending thread's queue.

In order to conserve processor resources according to an embodiment of the invention, the sending thread sends a signal to the receiving thread in conjunction with sending a message. The signal indicates to the receiving thread that a new message has been sent. This signaling may take the form of a Java script "send event" command. Signaling in this manner allows the receiving thread to simply check the queue when necessary, rather than constantly checking for incoming messages.

Referring to the flowchart of FIG. 4, as well as the diagram of FIG. 3, an example of how communication between two threads is initialized and conducted according to an embodiment of the invention will now be described. At step 200 (FIG. 4), thread 122 creates a message queue, labeled "Q1" (Item 126 in FIG. 3). At step 202, thread 122 spawns thread 124 and passes a reference to Q1 to thread 124 via the IDispatch interface 157. At step 204, thread 124 creates its own queue, labeled "Q2" (Item 128 in FIG. 3) and stores the reference to Q1 in a data structure labeled

"OtherQ" (Item 148 in FIG. 3). At step 208, thread 124 stores the reference to Q2 in another instance of "OtherQ" (Item 136 in FIG. 3). At step 210, thread 122 creates a message object (not illustrated) containing the text "HELLO." At step 212, thread 122 passes a reference to the message object – which, in most programming languages is simply the address of the object – to thread 124 for placement in Q2. The thread 122 then sends a signal to the thread 124 to indicate the presence of a new message at step 212. At step 214, thread 124 retrieves the message object 250 using the reference, and the process is complete.

An example of how an embodiment of the invention may be used in compiling a program will now be described with reference to FIG. 5. A server computer 160 is in communication with client computers 162, 164, 166 and 168, and console computer 180. Each client computer is responsible for compiling a section of a program. In this example, the program is divided into four sections: networking, kernel, user interface, and developer tools. These sections are compiled by client computers 162, 164, 166 and 168 respectively. The server computer 160 coordinates the activities of the client computers through the use of JScript commands. To accomplish this, the server computer executes a JScript engine 182 along five different threads of execution, labeled 170, 172, 174, 176 and 178. Consequently, each thread runs a copy of the JScript engine 182. Each copy includes one or more COM objects, such as COM object 184, for passing references to objects to other threads. Each of the threads 172, 174, 176 and 178 also includes a queue structured like the queues 126 and 128 of FIG. 3. Thread 170 includes a queue for each of the threads 172, 174, 176 and 178. In other words, thread 170 has a queue for

communicating with thread 172, a queue for communicating with thread 174, and so on. Threads 172, 174, 176 and 178 are responsible for executing scripting commands to control the compiling operations of client computers 162, 164, 166 and 168 respectively, while thread 170 is responsible for sending commands to coordinate the activities of the other threads. The types of commands sent by thread 170 include: "Start," for starting the execution of a thread; "Abort," for aborting the execution of a thread; and "Ignore error," to cause a thread to ignore an encountered error and continue its compiling operations.

The setup shown in FIG. 5 allows the thread 170 to send commands to each of the other threads in parallel. For example, it can send a "start" command to thread 172, and then send a "start" command to thread 174 without having to wait for thread 172 to have acted on its command. Once all of the "start" commands have been sent, the thread 170 can then update the status of the compiling operation on a user interface 182 of a console computer 180. This embodiment also helps solve problems related to the interdependency of the various sections of the program. For example, in a conventional compiling operation, the networking section of the program could not start compiling until the application programming interfaces (APIs) required for networking were defined in the kernel section. This would mean that no compiling could be accomplished on the networking section until the kernel section was complete. In the setup of FIG. 5, however, the networking section can be at least partially compiled in parallel with the kernel section until the APIs were needed.

It can thus be seen that a new a useful method and system for passing messages between threads has been provided. In view of the many possible

embodiments to which the principles of this invention may be applied, it should be recognized that the embodiments described herein with respect to the drawing figures is meant to be illustrative only and should not be taken as limiting the scope of invention. For example, those of skill in the art will recognize that the elements of the illustrated embodiments shown in software may be implemented in hardware and vice versa or that the illustrated embodiments can be modified in arrangement and detail without departing from the spirit of the invention. Therefore, the invention as described herein contemplates all such embodiments as may come within the scope of the following claims and equivalents thereof.